Next Up Previous Contents

Next: Crosscutting design paradigms **Up:** Operating System Concepts - **Previous:** The Design and Implementation **Contents**

Questions

1. **How is priority inheritence implemented ?** A very naive approach is this. Suppose a process P (with low priority) is going to get blocked and it has locked a resource R. The kernel needs to check if there are other higher priority processes that need that resource, if so then the priority of P is increased to that of the higher. Is this done ? PCB has place to maintain priority (it now has to maintain *real* and *effective* priority). Yes, this is how it is done, however, almost no flavors of Unix does priority inheritence.

2. **What is the advantage of kernel level threads over user level threads ?** - If there is a single kernel thread and there are multiple user level threads using that thread and if the kernel thread blocks doing I/O or something then none of the other user level threads can be scheduled as the entire process has to block. However, with kernel level threads if a kernel thread does a blocking system call then another kernel thread can be scheduled.

3. **In demand paging is it done directly from the file system or should there be enough space in the swap space?** If it is done directly from the file system then when a process is running we should not be able to modify the file system copy of the program. **No** this is incorrect. When the disk copy is removed (by doing an rm) the name no longer points to the inode. However, the incore copy of the inode still has a reference to it (because of the executable that is running) and only when the reference count is zero is the inode added to the free list of inodes on the disk. While the program is running and another copy of the program is compiled then the program will be stored with the same name but with different inode and hence there is no confusion. BSD Unix and Solaris 2 seem to use the approach that pages which can be kept in the file system i.e. the text part of a process, are kept in the file system for demand paging. However, the stack and data part of the process use the swap space for paging. Swap space generally uses much larger block size and does not involve indirection present in accessing the file system and is thus much faster than file system access. Another approach is to demand page pages directly from the file sytem when the pages are brought in for the first time, but upon replacement the pages are sent back to swap space thus ensuring that only previously used pages are placed in swap space and not the entire program.

4. **Can the kernel address space be paged ?** Yes, but there are also parts of the kernel that are wired (i.e. cannot be replaced by pageout daemon). The kernel space is generally a fixed part of virtual address and only the rest can be used for normal processes. This makes the kernel simpler because no matter how you enter the kernel (because of different process system calls or because of interrupts or what not) the kernel should have a consistent view of its address space.

5. **How does the kernel manage free frames and pages ?** The kernel has to keep a list of free frames in physical memory. It also has to maintain a list of the pages sorted according to some order (say LRU, etc) so that page eviction can be done.

6. **What is the page replacement algorithm used by FreeBSD and how is it implemented ? What policy is used for frame allocation between multiple processes ?** It does a modification of the second chance algorithm. Frames are scanned and if the frame is already free then it is

added to the free frames list. If it is not then the corresponding page table is found out (how does this work, back pointers stored ?). If the entry says that the entry is invalid then the page is added to the free list, but if the page is valid then it is made invalid but reclaimable (i.e. if it does not get paged out by the next time it is wanted, it can just be made valid again). FreeBSD wakes up the pagedeamon whenever the number of free frames becomes lower than *lotsfree* (typically 1/4th of memory that is used for paging). Entire processes are swapped out (instead of just pages being paged out) if the free frames become less than *minfree* and the average recent free pages goes below *desfree*. Atleast *minfree* pages are required to supply any pages required to handle interrupts.

7. **When a program such as 'ls' is run and an image of 'ls' is already in memory, how does the kernel maintain that the 'ls' program is already in memory ? This needs to be done both for fork() as well as for exec().** My guess is that the list of text structure (maintained always in kernel memory) is searched. This structure either points to the memory where the text is present or else points to the swap address where the text is swapped out. Of course the text might not be present at all. However, how is consistency maintained ? For example if I ran a version of 'ls' a few minutes ago and now have been able to recompile a new version of 'ls' (overwriting the earlier version of 'ls') then how does the kernel maintain consistency ? Is the text structure also used for searching for dynamically linked routines ? I think the consistency is maintained because the text structure list is searched using the inode rather than file name since file name could be the same for a more recent copy of the executable but the inode will have changed. Also, most lists maintained in the kernel can also be indexed using a hash function in order to directly reach a particular entry rather than having to traverse the entire list.

8. **How does the Unix Fast File System work ? What assumptions does it make for maintaining consistency of data ?** It uses something called a cylinder group. See section A.7.7. Can all the arms move independently or do they all have to be placed at the same cylinder ? No, the arms have to move in synchrony hence all arms read the same cylinder at a time.

9. **How are soft links and hard links implemented in Unix ?** Soft links are exactly as normal files (though the inode does mark it is a soft link) and the data that the inode points to is interpreted as a file name. Hard link on the other hand is just an entry into the directory structure which has the same inode as the linked file that it is pointed to. The inode of the file has to maintain a count of the number of links so that deletions can be handled correctly. Without a reference count how can the OS decide whether free the inode for the file or not.

10. **What does *fsck()* do ? How does the OS know to run fsck in the first place ?** OS knows to run fsck() in one of many ways. One probable way is to maintain a bit in the superblock. Whenever the machine boots up it checks that bit and if the bit is 0 then fsck is run, else the bit is set to 0. When the machine does a graceful shutdown then the bit is again set to 1. Thus, if there is a crash then the bit will be set to 0 when the machine boots up and hence fsck will be invoked. fsck() basically has to correct all the metadata information. I checked with the freeBSD 5.0 implementation and the superblock structure does have such a field. It has to check all data blocks and see to it that it either belongs to the free list or to an inode. Inodes should similarly be checked. The size of the file in the inode should also match with the data blocks that the inode points to. Hence, the data block should first be written and then the size of the file increased.

11. **How does the OS keep track of time ? How is timer interrupt implemented ?** There is a separate hardware clock which keeps on ticking at some regular interval. It can be loaded with a number and the clock starts counting down for that many ticks and when the counter reaches zero the clock sends an interrupt. The interrupt handler for this interrupt thus gets notification about the

elapsation of a time period and also reloads the number into the device register so that the clock can start counting down again. Modern day processors also allow access to the actual hardware clock used by the CPU to execute instructions, however that cannot be set and can only be read. **Is this clock the same as the CMOS clock as referred in adjkerntz() ?**

12. **How are cron jobs implemented ?** The daemon wakes up every minute and runs all processes that need to be run in that minute.

13. **Is double buffering used in freeBSD ?** I mean the producer writes in one buffer and then when it fills up starts writing to the second buffer. By the time that the second buffer fills up the first buffer should have been consumed by the consumer. However, this still requires some sort of synchronization since it is very easy for a fast producer to fill up both the buffers and the the consumer still consuming from the first buffer.

14. **In FreeBSD does the OS decide on disk access scheduling or does the disk controller decide or do both cooperate ?** In Section A.8.1 it is said that the device driver decides on the order of the writes, thus emphasizing that the OS does it. If this is so then how are bad sectors managed, by sector striping ? These days each vendor has proprietory technology to manage bad sectors and also to order accesses. i.e. there is no guarantee that the access pattern that the OS thinks is the best is actually the best because the disk has its own intelligence.

15. **What is the technology of a disk drive ?** I suppose they are magnetic since they allow such easy rewriting. Probably these are magneto-optic disks. The magnetic field is too weak to change the magnetic orientation under room temperature. A laser beam is used to heat up the bit position and then the same magnetic field can change the magnetic orientation. Reading is done by sending a laser beam and checking if the reflected beam has its poles rotated clockwise or anticlockwise.

16. **How are capabilities implemented ?** A file descriptor is a special case. How about generic capability based access permission ? An ACL is maintained as a 2D matrix with row being indexed by domains (and processese belong to a particular domain) and the columns being indexed by objects. The entries are read, write, execute, etc. The column also has entries for the domains so that switching from one domain to another can be controlled as well. The implementation of this is straightforward: either as a matrix or as separate lists for each column (object). For capability based access control there is a list maintained for each domain, i.e. we maintain the columns for each row (instead of the other way round for ACLs). However, **how can this make capability checking more efficient than ACL checking ?**

17. **What happens when a user is allowed to open a file to read and after the open is successful, the owner of the file takes away the read permission for the user ?** I wrote a program which opens a file with read permission and then waits for a ENTER. While it waits for the enter I revoke the read permission of the file. However, now when I continue the program the program is still able to read from the file! This demonstrates the revocation problem with capability. Let us see what are the steps involved. The inode has to be changed since the permissions are kept in the inode. Unix does metadata change synchronously. Thus, once the user process has permission to read the file the only place where the permissions are checked is the per process file table. **Can't this problem be solved by having to check the inode (the cached copy) if the inode has changed ?** Yes it can be done with the information that is present in the in memory inode, but that is not very efficient because then every access would have to go through the same user-group-others identification that has to be done when the file is first 'open'ed.

18. **What are the steps involved in translating a virtual address to physical address, involving L1 and L2 caches as well ?** See page 467 (Fig. 5.37) of Hennessy Patterson - Third Edition (in Summary of Virtual Memory and Caches).

19. **Which monitor is more commonly used ? Hoare's or the other (MESA)?** Mesa is more commonly used, a process switches between waiting for a shared variable (blocked state) or runnable. i.e. the original process completes execution inside the monitor before the awakened process can be run. MESA monitors are more commonly used (almost always) because there is only one context switch required because the signalling process runs to completion before the signalled process enters the monitor. In Hoare's monitor the signalled process had to run (the first context switch) and then context switch back to the signalling process.

20. **What are the various latencies (seek, rotational, transfer) of a commonly used disk ?** In milliseconds.

21. **How is information about free disk blocks maintained in Unix file system ?** See FFS in the Appendix on FreeBSD to Silberschatz's book.

22. **Does Unix use RAM disk ?** Not normally. However, FreeBSD does have a port available for RAM disk and it is in its standard distribution. See Section 1.10

23. **What is the difference between typing http://a.b.c and http://www.a.b.c ? Is it a standard feature ?** There is no standard feature other than the fact there have to be two entries in the DNS namespace, one each for a.b.c and www.a.b.c.

24. **How do you set hardware protection bit (for example for copy on write pages) ? Is it just a bit in the page table entry ?** Yes, it is simply a bit in the page table entry.

25. **How does a debugger work ?** It atleast changes the mode of the text segment from read only to read-write. This is done through the ptrace() system call in unix which allows a parent process to change the text of the debugged (child process) and set breakpoints. The debugger also uses symbol table information which are maintained normally in the disk image of the executable and are normally never used while running a process. **Figure 3.3 of Design and Implementation of 4.4 BSD has a nice diagram of the entire setup of memory of a process**. Each directly executable file has the following:
    1. A magic number saying whether the file can be paged and whether the text part of the file can be shared among multiple processes.
    2. Following the magic number is an *exec* header which specified size of text, initialized data, uninitialized data, and additional debugging information (debugging information is not used by the kernel or by the executing program; it is used by debuggers, say gdb).
    3. Following the header is the text, initialized data, and the symbol table (for debugger). The unitialized data is not kept in the disk since they can be generated on demand by zero-fill-on-demand memory.

    The ptrace() system call provides tracing and debugging facilities. It allows one process (the tracing process) to control another (the traced process). Most of the time, the traced process runs normally, but when it receives a signal (see sigaction(2)), it stops. The tracing process is expected to notice this via wait(2) or the delivery of a SIGCHLD sig- nal, examine the state of the stopped process, and cause it to terminate or continue as appropriate. The ptrace() system call is the mechanism by which all this happens. ptrace allows single stepping, reading and writing directly

into the virtual address space of the traced process and all other functions that we normally do with gdb. **See 'man ptrace'.**

Debuggers also use *private mapping* of the executable so that when the debugger sets a breakpoint it is not set in the file copy of the executable and is also not visible other processes using that same text. The kernel uses shadow objects to prevent this. The mappings are done as *copy-on-write* and thus when the debugger wants to modify the text, a page fault occurs and the traps into the kernel. The kernel then makes a copy of the page. **See Private Mapping Page 142.** Shadow pages are not written back to disk, they are simply freed when the process exits. When a page fault occurs the kernel first checks the shadow pages and then the other pages, thus ensuring that the copied page is returned instead of the original page.

When a breakpoint is set at a particular instruction the text of the program (the memory copy of it and not the disk image) is changed so that the instruction corresponding to the instruction is overwritten with a special "breakpoint" instruction. When the breakpoint instruction executes there is a trap generated and that sends a signal to the parent process (i.e. the process that is ptracing the child process) and the parent process can then do one of several things. If it chooses to step one instruction at a time then it again sets a breakpoint in the next instruction. Also, the debugger program does the bookkeeping so that the instruction that is showed to the user is the actual instruction and not the special breakpoint instruction.

26. **How is context switch implemented ?** All the information that needs to exist to run a program is present in the PCB. FreeBSD and most Unix implementations have separated the PCB into two different parts, one memory resident (always kept in memory) and the other which maybe swapped out. All information that is needed only when the process is executing can be kept in a structure (u area) which can be swapped out. Pids, gids, need to be in the in memory part so that processes can be killed (i.e. signals can be delivered).

27. **How does FreeBSD store page tables ? Why is there a need for zero-fill-on-demand pages ? i.e. what do we stand to loose if the contents are not zeroed out ?** The OS does not decide how to store page table. It is dictated by the hardware. The current intel architecture (Pentium 4) stores a hierarchical page table which has each part as large as a page. The page size is 4KB requiring 12 bits for offset within the page. Thus we are left with 32-12 = 20 bits to organize our page table and Pentium 4 splits it into equal chunks of 10 bits each, thus the first level has 2^{10} entries and each entry is a word = 4B and hence the entire first level table is $2^{10} \times 4B = 4KB$ = page size. The

page table has to be contiguous in physical memory.

However, what action is to be taken on a Page Table miss is dependent on the OS implementation. For example the *bss* of a program contains data that just needs to be zeroed out. Thus the initial entry in the page table is *invalid* and in the remaining bits of the 4B page table entry a bit pattern is stored. When the page table miss generates an interrupt because the page table entry is invalid, the handler checks what bit pattern is present in the page table entry, if it matches a certain pattern (say for bss pages) it knows that it can allocate any free page from physical memory to and simply zero them out and that should to it. This is what is called *zero-fill-on-demand* pages. Strictly speaking these pages can be left as is but most programs assume uninitialized data to be initialized to zero and hence programs start malfunctioning if the page allocated in this manner is not zeroed. If, however, the bit pattern says that the page refers to initialized data that is present in the disk image of the executable then the page has to be fetched from disk. Similarly, if the bit pattern refers to the executable itself then the handler will first search if that executable is currently being

used by some other process and if so then the OS looks into the text structure which must already be maintained for that executable and uses that to service the page fault.

See Page 134 for a description of what happens when there is a page table miss.

28. **In a hashed page table who implements the hash function, OS or hardware ?** I would be surprised if the hardware did it since hash functions are typically quite complicated. Also, if it were to be implemented in the hardware then the OS cannot change the hashing function, which again seems to suggest that the hash function needs to be implemented in the OS and not in the hardware.

29. **How is shell implemented ?** Is it as simple as vfork() and then exec() ? Yes. It could also be done as fork (instead of a vfork) followed by exec but vfork is faster than fork since it does not do any copies whereas fork has to do the copy of the user structure.

30. **Compilation stages of a program ?** Section 1.3 (Page 10) of dragon book (Aho, Sethi, and Ullman) gives a brief overview. The stages are as follows:

    1. Lexical analysis (parsing)
    2. Syntax analysis
    3. Semantic analyzer
    4. Intermediate code generator
    5. Code optimizer
    6. Code generator (assembly language output)
    7. Assembler (machine code output)
    8. Linker - loader

    At each stage the compiler uses the symbol table to keep information about symbols that the compiler encounters. Whenever a symbol is seen while parsing, the symbol table is checked for the scope, type, etc. of the symbol. If the symbol table does not have an entry for the symbol encountered then a new entry is made. Thus the table has to support fast searching and also insertions (and deletions).

31. **It seems that the system maintains whether a process did an exec after fork (Page 71, Section 3.9 of Design and Implementation of 4.4BSD). Why is this information maintained ?**

32. **What is the difference between the memory management information maintained in the process structure and the memory management information present in the PCB ? What is processor-state longword ?** For the first part, the process structure has to be able to identify whether the process is in the swap space or is in memory, that is one information that the process structure has to maintain in memory. Conceptually, all the information regarding a process is called the PCB. However, for efficiency, the PCB has been divided into two parts, one part which needs to be in memory all the time (such as PID, so that that process can be killed, signalling info for example whether a signal has been masked or not, etc). The rest of the PCB (called PCB in FreeBSD or BSD) can be moved to swap space if need be.

33. **What are process groups and sessions ? Just ways of grouping processes and associating a control terminal to a group of prosseses ?** A group of processes is called a *process group* whereas a group of process groups is called a *session*.

34. **Why does the TLB need to be flushed when a second mapping for an already mapped physical page is done - for a copy-on-write operation ?** This is obvious because if the TLB is not flushed then the virtual address will find a hit in the TLB and hence will update the physical page correspdonding to the virtual address and hence the copy-on-write semantics cannot be implemented. Only when there is a miss in the TLB is the page table consulted and then only can it be found that the page that the process is trying to access is a copy-on-write page and so it has to be copied before the page can be written upon. This is because the copy-on-write information is maintained in the page table and not in the TLB and hence the TLB has to have a miss before the page table is consulted.

35. **The a.out magic number says whether the executable can be paged and whether the text part of the file can be shared between multiple processes ?** It is there for historic reasons and these days it is the same magic number for all executables.

36. **When a process's parent dies it becomes the child of the init process; does the init process do a wait() on the child ? if so how does it know that it has a child ?** The init process keeps going over the process list and waits on the processes so that any orphan process can exit cleanly (because the parent has to do a wait before the process structure for a process can be freed).

37. **How is ioctl() used to add streams to write device driver code ?** This is not very used. It is a very easy way of adding layers of code before the device driver.

38. **How are semaphores implemented in Unix ?** You need some special instruction like some variant of *test-and-set*. You could also do this using *Load Linked* and *Store Conditional* instructions.

39. **How is private functions, etc. enforced ? Is similar method used to implement monitors ? How are monitors implemented in Unix ? For that matter how is scopping ensured ?** There has to similar protection mechanism like normal private functions of objects. The only protection is in what can be named and what cannot be named and hence all of these are ensured by the compiler. There is no protection other than that and if the compiler does not check for naming restrictions then the program can access private variables, etc. For monitors the compiler adds mutex codes for the entry procedure and the exit procedure.

40. **If memory is protected by such a simple thing called limit and relocation register then why are there security issues ?** In case of normal page tables (not hierarchical or inverted) the limit is used to check the size of the page table. However, for hierarchical page table or for inverted table, there is no such check. If an entry is not found then that is similar to having crossed the limit for normal page table.

    Buffer overflow happens as follows:

```
return address
local variable
....          <-- stack pointer
```

    The local variable can contain a buffer and if data can be written to the buffer without checking the bound of the buffer then the return address can be overwritten with some address which again points back to somewhere in the local data section and since most processors do not pay any heed to the executable permission (processors do heed the read and write permission) the buffer can be written with the code that needs to be executed. If the program that has been just taken over was

running as superuser then the machine can be overtaken (original Morris worm exploited the buffer overflow in finger and finger runs in kernel mode).

41. **How is *long long* or *quad* implemented in a 32 bit architecture ?** Either the compiler replaces multiple instructions for executing operations on long long or the library can implement such operations. The hardware has no idea what a long long is.

42. **How is CNTRL-C implemented so that the user can kill a process even before the entire input has been processed ?** The interrupt handler which gets called anyway because the characters enter the device driver using the interrupt handler. Since the handler is inside the kernel, the handler can understand special character such as CNTRL-C and then send a SIGKILL to the process.

43. **What is soft update ?** Metadata update in most file systems are synchronous. Log based filesystems use some sort of other information on the disk which helps to reconstruct the file system after a crash. The use of soft updates obviates most synchronous writes AND the need for a separate log. It tracks and enforces metadata update dependencies to ensure that the disk image is always kept consistent. This approach reduces disk writes by around 40-70% for file intensive environments (such as program development, mail servers, etc). Soft updates also provide better disk consistency. Soft updates ensure that the only inconsistencies are unclaimed blocks and inodes and thus file system check need not be started after every crash. Rather the machine can start up immediately and the unclaimed blocks and inodes can be recovered using a background procedure, thus increasing system usability. The performance approaches that of memory based file systems.

    Soft updates consider dependencies at the pointer level and not at block level. The basic idea is as follows: A dirty block can be written to disk at any time, as long as any updates within the in-memory block that have pending dependencies are first temporarily undone (rolled back). This guarantees that every block written to disk is consistent with the on-disk state. During the disk write, the in memory block is locked to prevent applications from seeing the rolled back state. When the disk write completes, any undone updates are restored back before the block is unlocked. For example:

```
Inode block             Directory block
=====================================
Inode #4 (free)         Free
Inode #5                B, #5

Suppose file A is added to the directory. Then the state is as follows:

Inode block             Directory block
=====================================
Inode #4 <-----------   A, #4
Inode #5                B, #5

Where the arrow head has to be done on disk before the tail of the arrow.

Now suppose we delete file B. Then the state is as follows:

Inode block             Directory block
=====================================
Inode #4 <-----------   A, #4
Inode #5 (free) ----->  free
```

All of the above metadata changes can by the following sequence:

1. Update directory block on disk to say that B has been deleted. A need not be added now.
2. Update the inode block in disk for file creation of file A.
3. Update the directory disk block in disk to include file A.

The good thing about this is that after each write the file system is in a consistent state and hence each of these writes can be done independently as well as in the background. Thus under the assumption that disk drives guarantee atomic writing to a block, this example can never create an inconsistent file system. If the computer crashes the disk need not be checked. However, there might be minor inconsistencies such as unclaimed inodes and data blocks. These can be handled by a background *fsck* process. The important thing is that it is guaranteed that the file system is consistent and *fsck* is needed purely for efficiency reasons.

Soft updates *does not* solve the persistency dangers that applications face. The only problem it solves is that it makes metadata update asynchronous.

44. **What does it mean for a directory to have its sticky bit set ?** A directory whose `sticky bit' is set becomes an append-only directory, or, more accurately, a directory in which the deletion of files is restricted. A file in a sticky directory may only be removed or renamed by a user if the user has write permission for the directory and the user is the owner of the file, the owner of the directory, or the super-user. This feature is usefully applied to directories such as /tmp which must be publicly writable but should deny users the license to arbitrarily delete or rename each others' files.

45. **How does signals work ? What is signal trampoline code?**

Signals can be posted by any process or by any code that runs at interrupt level. Normally signals are delivered in the context of the receiving process but if the signal forces the process to stop then the action can be taken immediately.

Signal implementation has two parts:

1. Posting a signal to a process

   When a signal is posted to a process it is added to the list of signals that have already been posted and are pending delivery. The process structure maintains a *p_siglist* that contains this list of undelivered signals. However, if the signal was being ignored then *p_siglist* need not be updated. When a signal is added to *p_siglist* it also does some implicit actions if there are any. For example if a SIGCONT is posted and there is a signal which would stop the process (SIGTTOU, SIGSTOP, etc) then those signals are removed from *p_siglist*. After this, if the process is masking the signal then the signal is ignored. If a process is being debugged (ptraced) then the parent process is always allowed to do intercede before the signal is delivered.

2. Recognizing the signal and delivering it to the target process.

   Each time that a process enters the system, it checks if it has pending signals and if so then either of the following two steps are taken:

   1. Produce a core dump
   2. Invoke a signal handler (specified by the process).

   The OS arranges for the signal handler to be called immediately after the process returns to

the user mode. This means that there has to be some special way since the process would normally follow the stack and that has no notion of a signal handler. This is done by the signal trampoline code. The user stack is modified so that on return to user mode, a call will be made immediately to a body of code (termed the *signal trampoline code*). This code invokes the signal handler, and if the handler returns, changes the process's signal state to the state that existed before the signal. *Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special signal stack.*

While delivering the signal a new sigmask is installed which is the union of the original sigmask, the signal being handled (so that there is no infinite recursive calls of the same signal handler), and any sigmask associated with the signal handler for the current signal.

46. **How does mmap() (i.e. memory mapped I/O) work ? How does the pager for a file know exactly what to fetch from disk ?** See Figure 5.7 in Design and Implementation of 4.4BSD.

47. **What is the structure of a *vnode* ?** Not at all important. Page 219 in Design and Implementation of 4.4BSD.

48. **What does it mean for a file to have "S" , that is the setuid bit is set but the file is not executable ? Similarly for "T" and "t" (only for "others" permission).** It does not mean anything. Not sure why such an option needs to be there.

49. **How does select work ?**

    When a process does a select, the select routine corresponding to the device is called. If it returns true then the select can return. Returning true does not mean that there is data to read, it means that a read will not block. If select routine of the device does not return true then the device select routine is responsible for remembering that the particular process was trying to do I/O. When I/O becomes possible for the descriptor, generally due to an interrupt in the underlying device, a notification is sent to the selecting process. The select routine is not able to remember multiple process ids and hence if multiple processes are selecting on the same descriptor then a *collision* occurs and notification is sent to all processes which are waiting for any descriptor.

50. **Describe the different stages when a process makes a system call ? (explain the kernel stack and user stack and how the transfer of control takes place between user space and kernel space)**

    System calls are a special form of software-initiated trap - the machine instruction used to initiate a system call typically causes a hardware trap that is handled specially by the kernel.

    The OS maintains a separate kernel stack when it goes into kernel mode (for a system call) so that when the kernel returns to the user the user cannot access the stack that the kernel used. Conceptually, the kernel could have just extended the user stack and could have used it but if such were the case then the data that the kernel wrote onto the stack would still be present and the user process can read those information since those information is still present in the memory just above the current user stack. Of course if that memory has been replaced due to memory management then there is no problem but the kernel cannot depend on memory management to actually ensure that. Thus to make things secure, yet simple, the kernel uses a separate stack which is in the kernel part of virtual memory, thus inaccessible by any user process.

51. **What is difference between non blocking I/O and asynchronous I/O ?** In the former the call returns with whatever data was read (or written) whereas in the later the call gets completed in its entirety but at a later time and the application is notified (asynchronously) when the call completes. Both have the property of being 'non-blocking' since the control returns immediately to the application after the system call.

52. **What is copy semantics ?** Suppose an application wants to write a buffer of data to disk. It does a write() system call with the address of the buffer and number of bytes that it wants to write. After the sytem call returns the application can change the buffer. Copy semantics says that the version of the data written to the disk is guaranteed to be the version that was present in the buffer at the time when the system call was made, irrespective of whether the buffer was subsequently changed or not.

53. **Use of spooling ?** Spooling is useful for devices that cannot handle simultaneous inputs from multiple applications (such as printers, tape drives, etc). It is a higher level of buffering.

54. **How does writing to a device work since Unix has a unified view of files and devices ?** See Page 479 (Section 13.5) of Silberschatz's OS book (6th Edition).

55. **What is used in FreeBSD for deadlock ? Prevention, avoidance or detection ?**

56. **Is there any protection associated with physical pages ?**

57. **How is linux compatibility done in FreeBSD ?**

58. **libc system call handling ?**

Next Up Previous Contents

Next: Crosscutting design paradigms **Up:** Operating System Concepts - **Previous:** The Design and Implementation **Contents**

Amit Kumar Saha 2004-03-03